UNIT – IV
ALGORITHMS

What is an Algorithm?

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Characteristics of an Algorithm

An algorithm should have the following characteristics −

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

Step 1 − START
Step 2 − declare three integers **a**, **b** & **c**
Step 3 − define values of **a** & **b**
Step 4 − add values of **a** & **b**
Step 5 − store output of step 4 to **c**
Step 6 − print **c**
Step 7 – STOP

Pseudocode conventions

We use the following conventions in our pseudocode.

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces: **{and}.** Compound statement can be represented as a block.
3. An identifier begins with a letter. The data types of variables are not explicitly declared.
4. Assignment of values to variables is done using the assignment statement.

Variable:= expression;

5. There are two Boolean values **true** and **false.** In order to produce these values , the logical operators and, or, and not and the relational operators <,≤, ≠, >, ≥ and =.
6. Elements of multidimensional arrays are accessed using[and]. Array indices start at zero.
7. The looping statements **for, while, and repeat until.**

```
while (condition) do
{
        Statement 1;
        .
        .
        .
        Statement n;
}
```

As long as condition is **true,** the statements get executed. When condition becomes **false,** the loop is exited.

The general form of a **for** loop is

```
for variable:=value1 to value2 step step do
{
        Statement 1;
        .
        .
        .
        Statement n;
}
```

here value1, value2, and step are arithmetic expressions. A variable of type integer or real or a numeric constant is a simple form of an arithmetic expression. The clause "**step** step" is optional and taken as +1 if it does not occur.

The general form of **repeat-until** is

```
repeat
    {
        Statement 1;
        .
        .
        .
        Statement n;
    } until (condition)
```

As long as condition is **false,** the statements get executed. When condition becomes **true,** the loop is exited.

8. **Break** can be used within any of the above looping instructions to force exit. A **return** statement within any of the above also will result in exiting the loops. A **return** statement results in the exit of the function itself.
9. A conditional statement has the following forms:

```
if(condition) then
statement;
if(condition) then
statement1;
else
statement2;
```

The following **case** statement:

```
case
{
    : (condition 1): statement1;
```

```
                .
                .
                .
            : (condition n): statement n;
            : else: statement n+1;
        }
```

10. Input and output are done using the instructions **read** and **write.**
11. There is only one type of procedure: **Algorithm.** An algorithm consists of a heading and a body. The heading takes the form

    Algorithm Name (parameter list)

    Where Name is the name of the procedure and parameter list is a listing of the procedure parameters. The body has one or more statements enclosed within braces **{ and }.**

```
Algorithm Max(A,n)
// A is an array of size n.
{
        Result:=A[1];
        for i:=2 to n do
                If A[i] > Result then
                        Result:=A[i];
        return Result;
}
```

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following −

- ***A Priori* Analysis** − This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- ***A Posterior* Analysis** − This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

Algorithm Complexity

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I.

Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)
Step 1 - START
Step 2 - C ← A + B + 10
Step 3 - Stop

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes **n** steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

Analyzing Algorithms

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

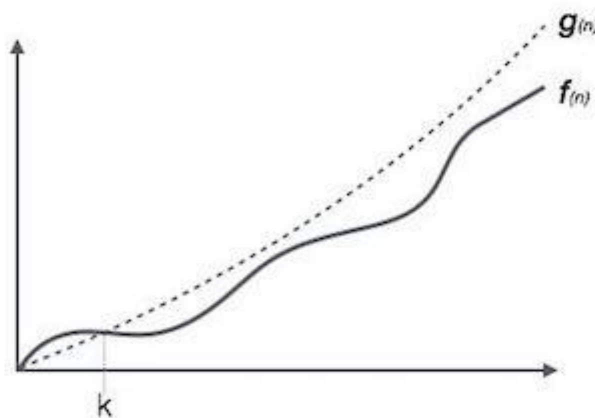
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- **O Notation**
- **Ω Notation**
- **θ Notation**

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

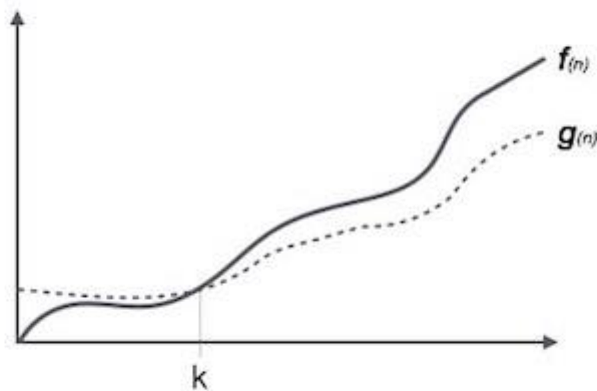


For example, for a function ***f*(n)**

$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq c.g(n) \text{ for all } n > n_0.$
$\}$

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

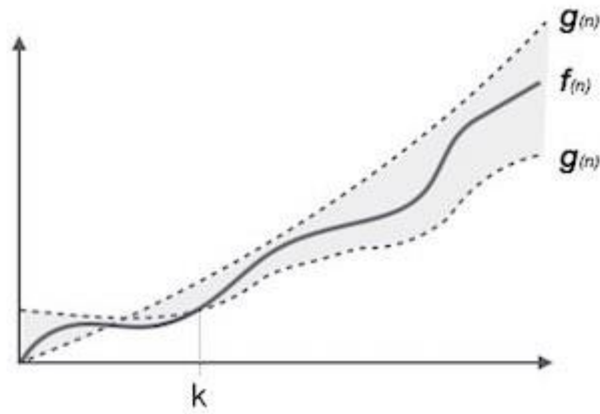


For example, for a function ***f*(n)**

$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c.f(n) \text{ for all } n > n_0.$
$\}$

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



$\theta(f(n))$ = { $g(n)$ if and only if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$ for all $n > n_0$.
}

Common Asymptotic Notations

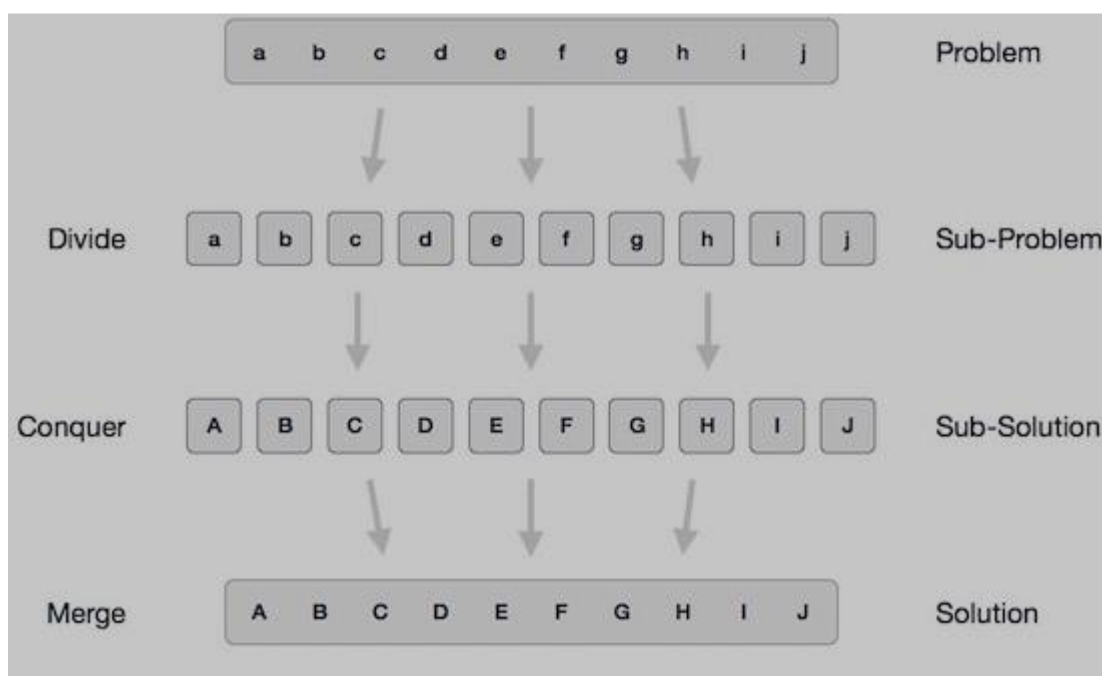
Following is a list of some common asymptotic notations −

constant	−	O(1)
logarithmic	−	O(log n)
linear	−	O(n)
n log n	−	O(n log n)
quadratic	−	$O(n^2)$
cubic	−	$O(n^3)$

polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$

Divide and Conquer

1. Divide the problem into number of smaller units called sub-problems.
2. Conquer (Solve) the sub-problems recursively.
3. Combine the solutions of all the sub-problems into a solution for the original problem.



Broadly, we can understand **divide-and-conquer** approach in a three-step process.

Divide/Break

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

Conquer/Solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Examples

The following computer algorithms are based on **divide-and-conquer** programming approach −

- Binary Search
- Finding the maximum and minimum
- Merge Sort
- Quick Sort
- Heap Sort
- Selection sort
- Strassen's Matrix Multiplication

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

Binary Search

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- Binary Search Algorithm can be applied only on **Sorted arrays**.
- There is a linear array 'a' of size 'n'. a_i, $1 \leq i \leq n$, be a list of elements that are sorted in nondecreasing order.
- Binary search algorithm is being used to search an element 'x' in this linear array.
- Let $P=(n,a_i,\ldots a_l,x)$ denote an arbitrary instance of this search problem.
- If search ends in success, it sets loc to the index of the element otherwise it sets position to -1.
- It works on the principle of divide and conquer technique.
- If P has more than one element, it can be divided into a new sub problem as follows. Pick an index q and compare x with a_q
- Variables low and end keeps track of the index of the first and last element of the array or sub array in which the element is being searched at that instant.
- Variable mid keeps track of the index of the middle element of that array or sub array in which the element is being searched at that instant.
- There are three possibilities:

  i. $x=a_q$: in this case the problem P is immediately solved.

  ii. $x<a_q$: in this case x has to be searched only in the sublist $a_i,a_{i+1},\ldots a_{q-1}$. Therefore, P reduces to$(q-i,a_i,\ldots,a_{q-1},x)$

iii. $x>a_q$: in this case x has to be searched only in the sublist $a_{q+1},\ldots a_l$. Therefore, P reduces to$(l-q, a_{q+1},\ldots a_l,x)$

```
Algorithm BinSearch(a,n,x)
{
    low:=1;
    high:=n;
    while(low≤high) do
    {
        mid:=(low+high)/2;
        if(x<a[mid] then
        high:=mid-1;
        else if(x>a[mid]) then
        low:=mid+1;
        else
        return mid;
    }
    return 0;
}
```

Time Complexity Analysis-

Binary Search time complexity analysis is done below-

- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.

Thus, we have-

Time Complexity of Binary Search Algorithm is $O(\log_2 n)$.

Here, n is the number of elements in the sorted linear array.

Binary Search Example-

Consider-

- We are given the following sorted linear array.
- Element 15 has to be searched in it using Binary Search Algorithm.

3	10	15	20	35	40	60
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

Binary Search Example

Binary Search Algorithm works in the following steps-

Step-01:

To begin with, we take low=0 and high=6. We compute location of the middle element as-

$$mid = (low + high) / 2 = (0 + 6) / 2 = 3$$

Here, $a[mid] = a[3] = 20 \neq 15$ and $low < high$. So, we start next iteration.

Step-02:

Since $a[mid] = 20 > 15$, so we take $high = mid - 1 = 3 - 1 = 2$ whereas low remains unchanged. We compute location of the middle element as-

$$mid = (low + high) / 2 = (0 + 2) / 2 = 1$$

Here, a[mid] = a[1] = 10 ≠ 15 and low < end. So, we start next iteration.

Step-03:

Since a[mid] = 10 < 15, so we take low = mid + 1 = 1 + 1 = 2 whereas end remains unchanged. We compute location of the middle element as-

mid= (low + high) / 2= (2 + 2) / 2= 2

Here, a[mid] = a[2] = 15 which matches to the element being searched. So, our search terminates in success and index 2 is returned.

Binary Search Algorithm Advantages-

The advantages of binary search algorithm are-

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

Binary Search Algorithm Disadvantages-

The disadvantages of binary search algorithm are-

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor.

Finding the Maximum and Minimum:

1. Let us consider simple problem that can be solved by the divide-and conquer technique.
2. The problem is to find the maximum and minimum value in a set of ‘n’ elements.
3. By comparing numbers of elements, the time complexity of this algorithm can be analyzed.
4. Hence, the time is determined mainly by the total cost of the element comparison.

```
Algorithm straight MaxMin (a, n, max, min)
// Set max to the maximum & min to the minimum of a [1: n]
{
Max = Min = a [1];
For i = 2 to n do
{
If (a [i] > Max) then Max = a [i];
If (a [i] < Min) then Min = a [i];
}}
```

Explanation:

a. Straight MaxMin requires 2(n-1) element comparisons in the best, average & worst cases.

b. By realizing the comparison of a [i]max is false, improvement in a algorithm can be done.

c. Hence we can replace the contents of the for loop by, If (a [i]> Max) then Max = a [i]; Else if (a [i]< 2(n-1)

d. On the average a[i] is > max half the time, and so, the avg. no. of comparison is 3n/2-1.

A Divide and Conquer Algorithm for this problem would proceed as follows:

a. Let P = (n, a [i],……,a [j]) denote an arbitrary instance of the problem.

b. Here 'n' is the no. of elements in the list (a [i],….,a[j]) and we are interested in finding the maximum and minimum of the list.

c. If the list has more than 2 elements, P has to be divided into smaller instances.

d. For example, we might divide 'P' into the 2 instances, P1=([n/2],a[1],……..a[n/2]) & P2= (n-[n/2], a[[n/2]+1],….., a[n]) After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

Algorithm:

```
MaxMin (i, j, max, min)
 // a [1: n] is a global array, parameters i& j are integers, 1<= i<=j <=n. The effect is
to4.
// Set max & min to the largest & smallest value 5 in a [i: j], respectively.
{
If (i=j) then Max = Min = a[i];
Else if (i=j-1) then
{
if (a[i] < a[j]) then
        {
                Max = a[j];
                Min = a[i];
        }
        Else
        {
                Max = a[i];
                Min = a[j];
        }
}Else
{
Mid = (i + j) / 2;
MaxMin (I, Mid, Max, Min);
MaxMin (Mid +1, j, Max1, Min1);
If (Max < Max1) then Max = Max1;
If (Min > Min1) then Min = Min1;
}}
The procedure is initially invoked by the statement, MaxMin (1, n, x, y)
```

Example:

A	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Values	22	13	-5	-8	15	60	17	31	47

Tree Diagram:

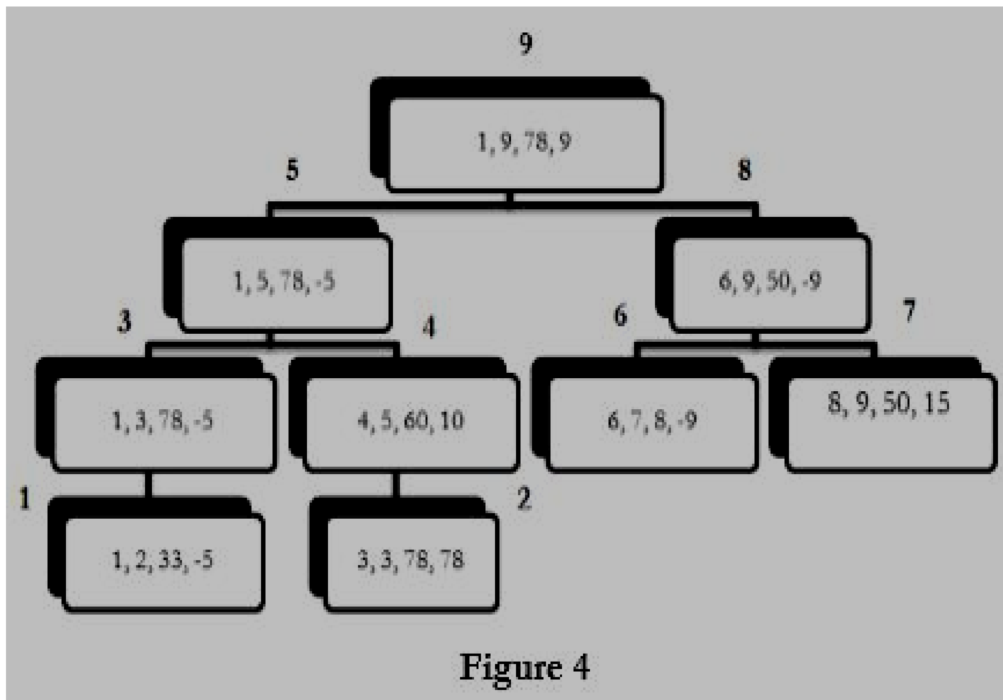


Figure 4

i. As shown in figure 4, in this Algorithm, each node has 4 items of information: i, j, max & min.

ii. In figure 4, root node contains 1 & 9 as the values of i& j corresponding to the initial call to MaxMin.

iii. This execution produces 2 new calls to MaxMin, where i& j have the values 1, 5 & 6, 9 respectively & thus split the set into 2 subsets of approximately the same size.

iv. Maximum depth of recursion is 4.

Complexity:

If T(n) represents this no., then the resulting recurrence relations is

T (n)=T([n/2]+T[n/2]+2 n>2

1 n=2

1 n=1

When ‘n’ is a power of 2, n=2k for some positive integer ‘k’, then

$$T(n) = 2T(n/2) + 2$$

$$= 2(2T(n/4)+2)+2$$

$$= 4T(n/4)+4+2$$

$$= 2k\text{-}1\ T(2) + \Sigma\ 1 \leq I \leq k\text{-}1 \leq 2i$$

$$= 2k\text{-}1 + 2k - 2$$

$$T(n) = (3n/2) - 2$$

Note that (3n/2) - 2 is the best-average and worst-case no. of comparisons when ‘n’ is a power of 2.

Heap Sort

Heaps can be used in sorting an array. In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array.

Consider an array Arr which is to be sorted using Heap Sort.

- Initially build a max heap of elements in Arr.
- The root element, that is Arr[1], will contain maximum element of Arr. After that, swap this element with the last element of Arr and heapify the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.
- Repeat the step 2, until all the elements are in their correct position.

Implementation:

```
void heap_sort(int Arr[ ])
{
```

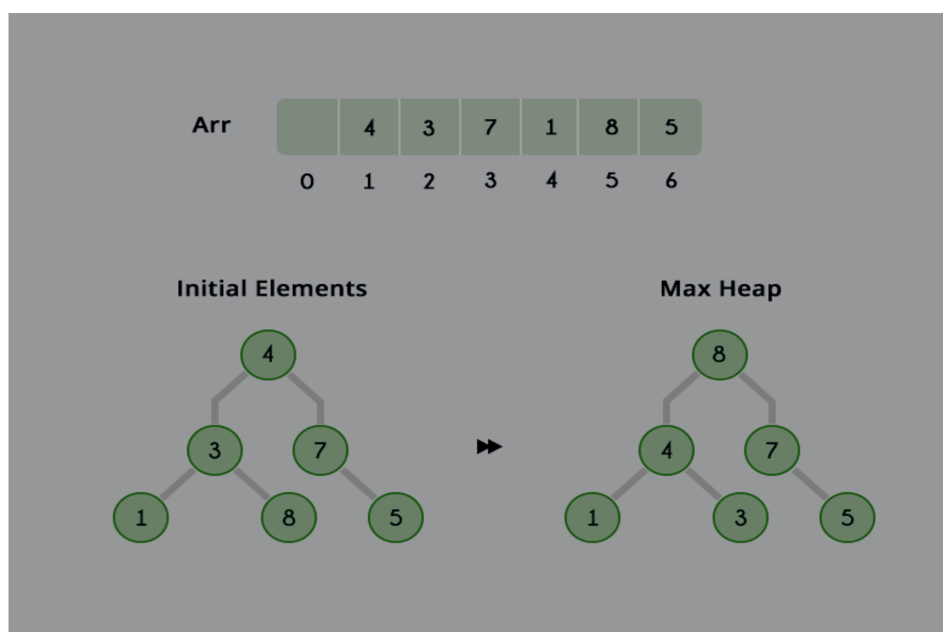
```
    int heap_size = N;
    build_maxheap(Arr);
    for(int i = N; i >= 2 ; i-- )
    {
      swap|(Arr[ 1 ], Arr[ i ]);
      heap_size = heap_size - 1;
      max_heapify(Arr, 1, heap_size);
    }
}
```

Complexity:

max_heapify has complexity O(logN), build_maxheap has complexity O(N) and we run max_heapify N−1 times in heap_sort function, therefore complexity of heap_sort function is O(NlogN).

Example:

In the diagram below, initially there is an unsorted array Arr having 6 elements and then max-heap will be built.



After building max-heap, the elements in the array Arr will be:

Arr		8	4	7	1	3	5
	0	1	2	3	4	5	6

Step 1: 8 is swapped with 5.

Step 2: 8 is disconnected from heap as 8 is in correct position now and.

Step 3: Max-heap is created and 7 is swapped with 3.

Step 4: 7 is disconnected from heap.

Step 5: Max heap is created and 5 is swapped with 1.

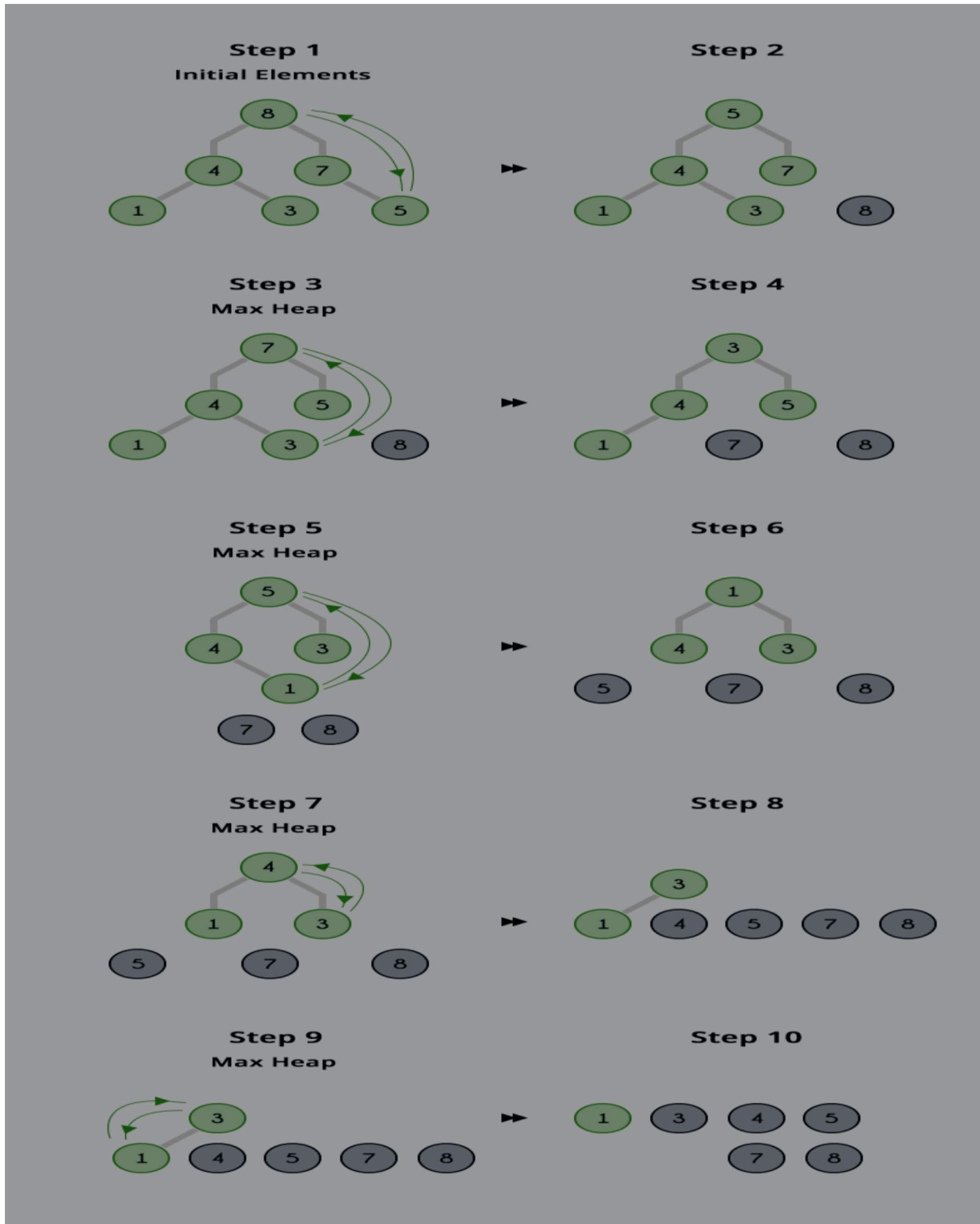
Step 6: 5 is disconnected from heap.

Step 7: Max heap is created and 4 is swapped with 3.

Step 8: 4 is disconnected from heap.

Step 9: Max heap is created and 3 is swapped with 1.

Step 10: 3 is disconnected.



After all the steps, we will get a sorted array.

Arr		1	3	4	5	7	8
	0	1	2	3	4	5	6

Merge Sort

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

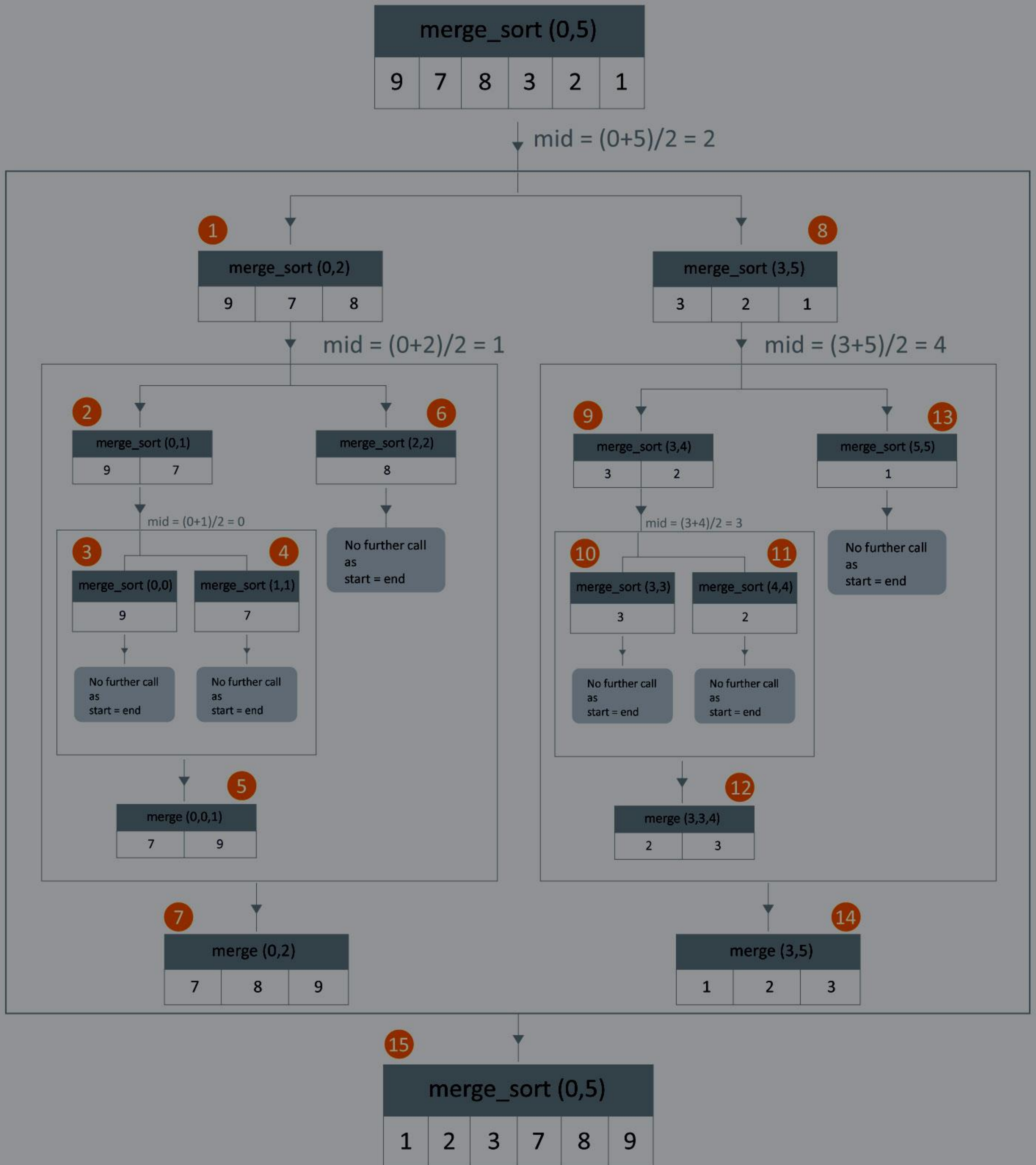
Idea:

- Divide the unsorted list into N sublists, each containing 1 element.
- Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.
- Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

Let's consider the following image

Merge Sort



- As one may understand from the image above, at each step a list of size M is being divided into 2 sublists of size M/2, until no further division can be done. To understand better, consider a smaller array A containing the elements (9,7,8).
- At the first step this list of size 3 is divided into 2 sublists the first consisting of elements (9,7) and the second one being (8). Now, the first list consisting of elements (9,7) is further divided into 2 sublists consisting of elements (9) and (7) respectively.
- As no further breakdown of this list can be done, as each sublist consists of a maximum of 1 element, we now start to merge these lists. The 2 sub-lists formed in the last step are then merged together in sorted order using the procedure mentioned above leading to a new list (7,9). Backtracking further, we then need to merge the list consisting of element (8) too with this list, leading to the new sorted list (7,8,9).

An implementation has been provided below :

```
 void merge(int A[ ] , int start, int mid, int end) {
 //stores the starting position of both parts in temporary variables.
int p = start ,q = mid+1;

int Arr[end-start+1] , k=0;

for(int i = start ;i <= end ;i++) {
   if(p > mid)      //checks if first part comes to an end or not .
      Arr[ k++ ] = A[ q++] ;

  else if ( q > end)   //checks if second part comes to an end or not
```

```
      Arr[ k++ ] = A[ p++ ];

  else if( A[ p ] < A[ q ])     //checks which part has smaller element.
     Arr[ k++ ] = A[ p++ ];

  else
     Arr[ k++ ] = A[ q++];
 }
 for (int p=0 ; p< k ;p ++) {
  /* Now the real array has elements in sorted manner including both
       parts.*/
    A[ start++ ] = Arr[ p ] ;
  }
}
```

Here, in merge function, we will merge two parts of the arrays where one part has starting and ending positions from start to mid respectively and another part has positions from mid+1 to the end.

A beginning is made from the starting parts of both arrays. i.e. p and q. Then the respective elements of both the parts are compared and the one with the smaller value will be stored in the auxiliary array (Arr[ ]). If at some condition ,one part comes to end ,then all the elements of another part of array are added in the auxiliary array in the same order they exist.

Now consider the following 2 branched recursive function :

```
  void merge_sort (int A[ ] , int start , int end )
  {
         if( start < end ) {
```

```
        int mid = (start + end ) / 2 ;          // defines the current array in 2 parts
.
        merge_sort (A, start , mid ) ;              // sort the 1st part of array .
        merge_sort (A,mid+1 , end ) ;            // sort the 2nd part of array.

      // merge the both parts by comparing elements of both the parts.
       merge(A,start , mid , end );
  }
}
```

Time Complexity:

The list of size N is divided into a max of logN parts, and the merging of all sublists into a single list takes O(N) time, the worst case run time of this algorithm is O(NLogN).

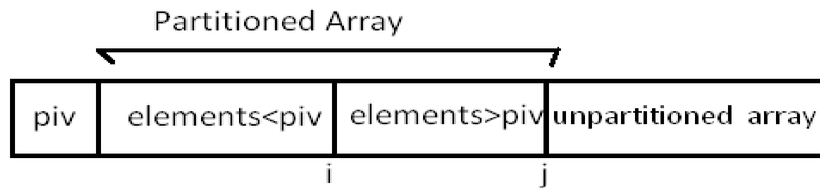
Quick Sort

Quick sort is based on the divide-and-conquer approach based on the idea of choosing one element as a pivot element and partitioning the array around it such that: Left side of pivot contains all the elements that are less than the pivot element Right side contains all elements greater than the pivot

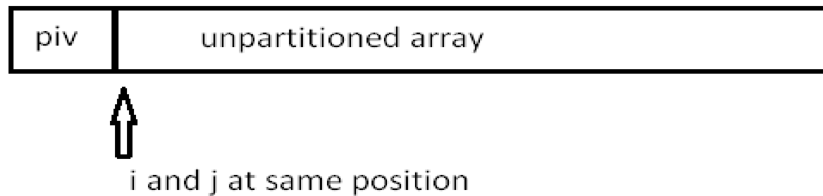
It reduces the space complexity and removes the use of the auxiliary array that is used in merge sort. Selecting a random pivot in an array results in an improved time complexity in most of the cases.

Implementation :

Select the first element of array as the pivot element First, we will see how the partition of the array takes place around the pivot.



Initially :



In the implementation below, the following components have been used:

Here, A[] = array whose elements are to be sorted

start: Leftmost position of the array

end: Rightmost position of the array

i : Boundary between the elements that are less than pivot and those greater than pivot

j : Boundary between the partitioned and unpartitioned part of array

piv: Pivot element

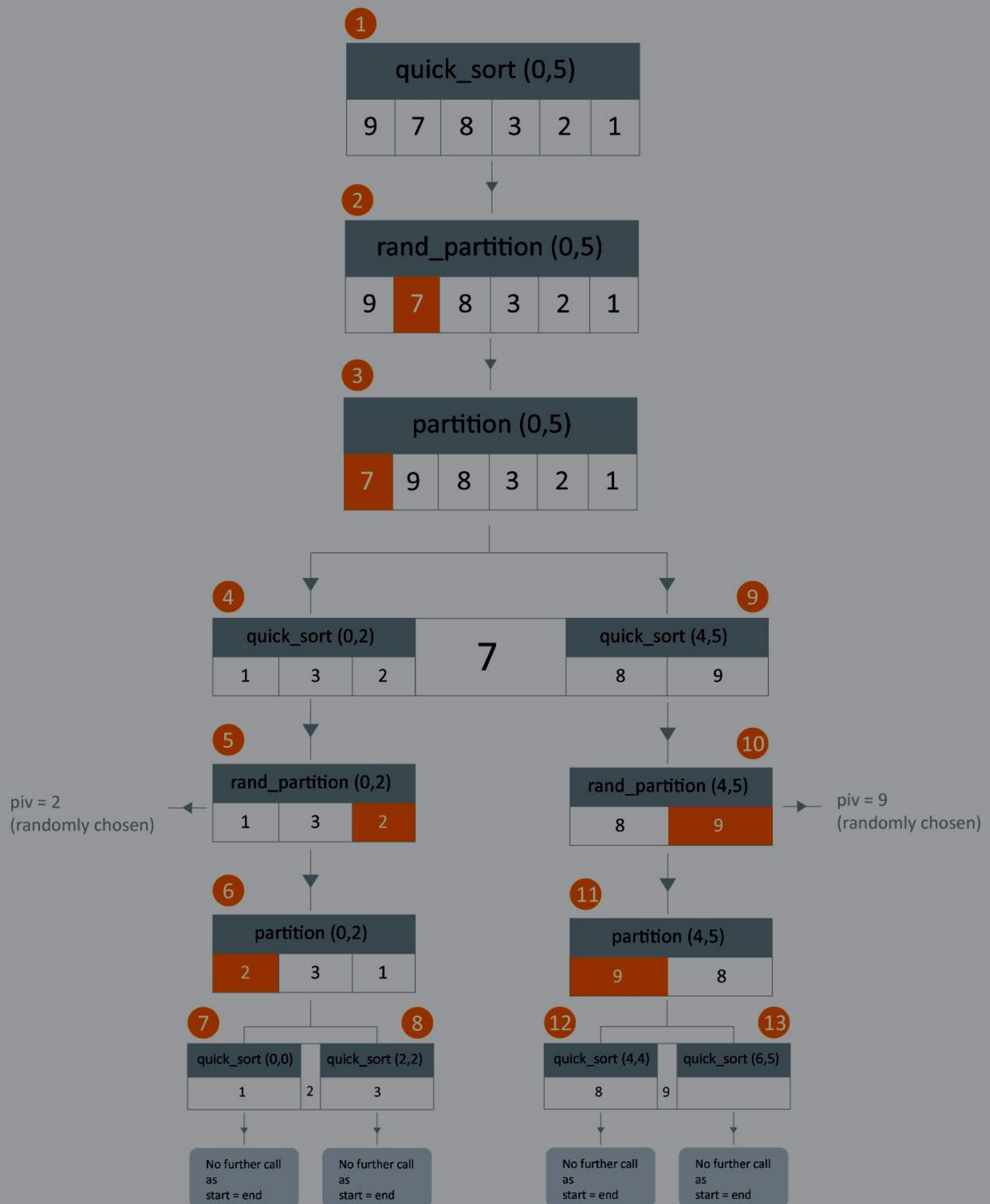
```
int partition ( int A[],int start ,int end) {
    int i = start + 1;
    int piv = A[start] ;            //make the first element as pivot element.
    for(int j =start + 1; j <= end ; j++ )  {
    /*rearrange the array by putting elements which are less than pivot
       on one side and which are greater that on other. */

          if ( A[ j ] < piv) {
                 swap (A[ i ],A [ j ]);
            i += 1;
        }
```

```
  }
  swap ( A[ start ] ,A[ i-1 ] ) ;  //put the pivot element in its proper place.
  return i-1;                      //return the position of the pivot
}
```

Now, let us see the recursive function Quick_sort :

```
void quick_sort ( int A[ ] ,int start , int end ) {
  if( start < end ) {
      //stores the position of pivot element
       int piv_pos = partition (A,start , end ) ;
       quick_sort (A,start , piv_pos -1);    //sorts the left side of pivot.
       quick_sort ( A,piv_pos +1 , end) ; //sorts the right side of pivot.
  }
}
```



Here we find the proper position of the pivot element by rearranging the array using partition function. Then we divide the array into two halves left side of the pivot (elements less than pivot element) and right side of the

pivot (elements greater than pivot element) and apply the same step recursively.

Example: You have an array A=[9,7,8,3,2,1] Observe in the diagram below, that the randpartition() function chooses pivot randomly as 7 and then swaps it with the first element of the array and then the partition() function call takes place, which divides the array into two halves. The first half has elements less than 7 and the other half has elements greater than 7. For elements less than 7, in 5th call, randpartition() function chooses 2 as pivot element randomly and then swap it with first element and call to the partition() function takes place. After the 7th and 8th call, no further calls can take place as only one element left in both the calls. Similarly, you can observe the order of calls for the elements greater than 7.

```
Let's see the randomized version of the partition function :
int rand_partition ( int A[ ] , int start , int end ) {
   //chooses position of pivot randomly by using rand() function .
    int random = start + rand( )%(end-start +1 ) ;

     swap ( A[random] , A[start]) ;        //swap pivot with 1st element.
    return partition(A,start ,end) ;       //call the above partition function
}
```

Use randpartition() instead of partition() function in quicksort() function to reduce the time complexity of this algorithm.

Time Complexity

The worst case time complexity of this algorithm is O(N2) , but as this is randomized algorithm, its time complexity fluctuates between O(N2) and O(NlogN) and mostly it comes out to be O(NlogN).

Selection Sort

The Selection sort algorithm is based on the idea of finding the minimum or maximum element in an unsorted array and then putting it in its correct position in a sorted array.

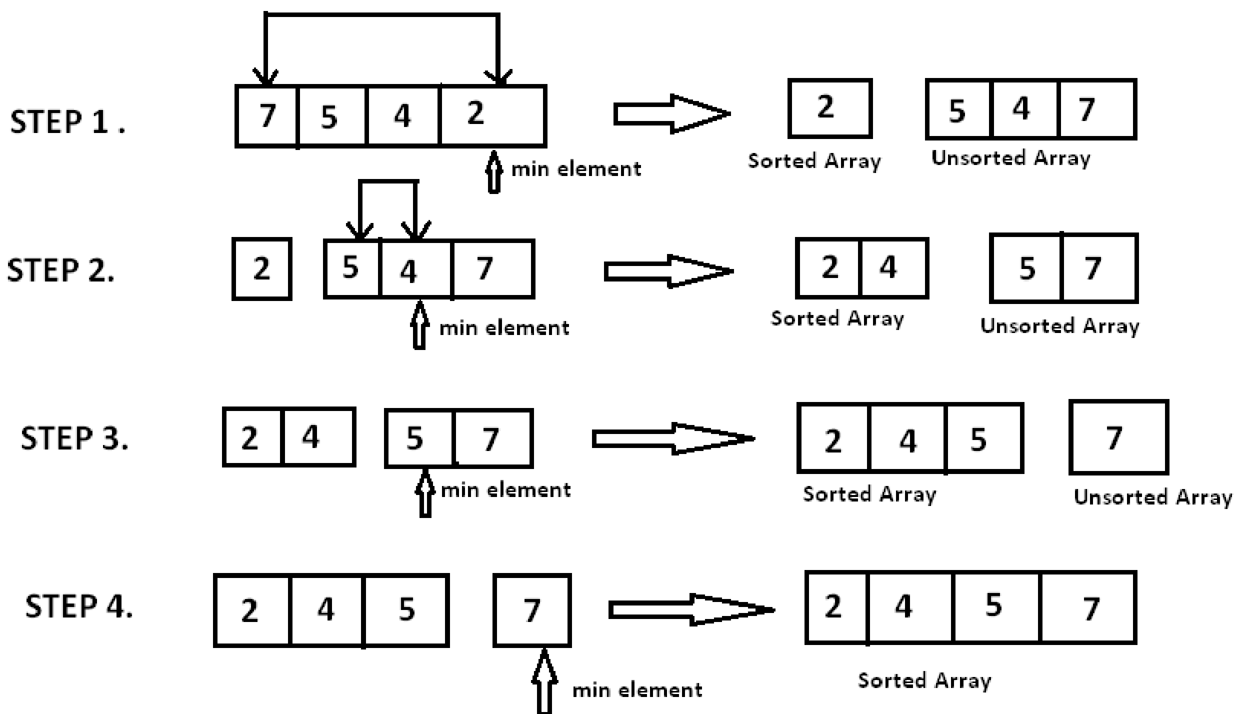
Assume that the array A=[7,5,4,2] needs to be sorted in ascending order.

The minimum element in the array i.e. 2 is searched for and then swapped with the element that is currently located at the first position, i.e. 7. Now the minimum element in the remaining unsorted array is searched for and put in the second position, and so on.

Let's take a look at the implementation.

```
void selection_sort (int A[ ], int n) {
        // temporary variable to store the position of minimum element
        int minimum;
        // reduces the effective size of the array by one in  each iteration.
        for(int i = 0; i < n-1 ; i++)  {
           // assuming the first element to be the minimum of the unsorted array .
             minimum = i ;
          // gives the effective size of the unsorted  array .
            for(int j = i+1; j < n ; j++ ) {
                if(A[ j ] < A[ minimum ])  {                //finds the minimum element
                minimum = j ;
                }
             }
          // putting minimum element on its proper position.
          swap ( A[ minimum ], A[ i ]) ;
        }
    }
```

At ith iteration, elements from position 0 to $i-1$ will be sorted.



Time Complexity

To find the minimum element from the array of N elements, $N-1$ comparisons are required. After putting the minimum element in its proper position, the size of an unsorted array reduces to $N-1$ and then $N-2$ comparisons are required to find the minimum in the unsorted array.

Therefore $(N-1) + (N-2) + ... + 1 = (N\cdot(N-1))/2$ comparisons and N swaps result in the overall complexity of O(N2).